Date __/__/____

Graph

Saathi

Graph Data Structure is a collection of nodes connected by edges. It's used to represent relationships b/w different entities. Graph algorithms are methods used to manipulate and analyze graphs, solving various problems like finding the shortest path or detecting cycles.

Graph is a non-linear data structure consisting of vertices & edges. The vertices are sometimes also refferred to as nodes & the edges are lines or arcs that connect two nodes.

Graph is composed with a set of vertices (V) set of edges (E). The graph is denoted by G(V, E)

Components of Graph:

(1) Vertices: Fundamental unit of graph. also known as vertex or nodes. They can be labeled or unlabeled.

(2) Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. sometime edges are also known as arcs. even edges can be labeled / unlabeled.

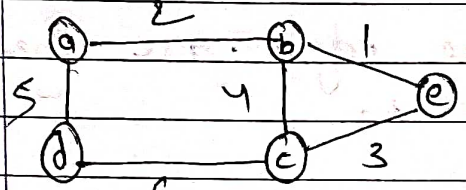
Date ___/___/_____

Saathi

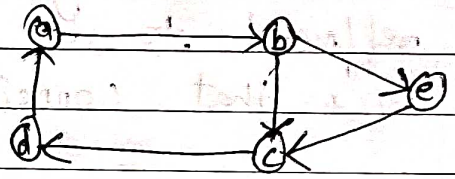
Basic Operations.

1. Insertion
2. Deletion
3. Searching
4. Traversal.

1. Basics of Graphs.
2. BFS & DFS in Graphs
3. Cycles in Graph
4. Shortest path
5. Minimum spanning Trees
6. Topological sorting
7. Connectivity in Graph.
8. Maximum flow in Graph



Undirected Graph



Directed Graph

a - b } only possible in undirected
b - a }

a - b ✓
b - a ✗

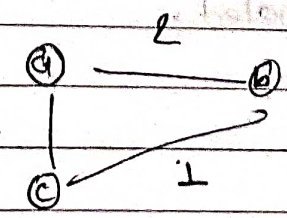
* Degree → Edges connected to a Node is the degree of Node. (for undirected)

* For Directed Graph.

(i) Indegree → Edges coming towards a Node.
Indegree of (b) = '1'

(ii) Outdegree → Edges going outwards from a Node
outdegree of (b) = '2'

* Weighted Graph → undirected →
if no weight is given assume '1'



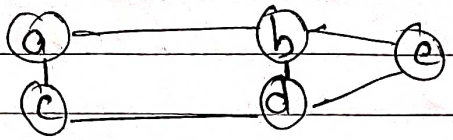
a - c → 2
a - b → 3
b - c → 1

Page No.

Date ___/___/_____

Saathi

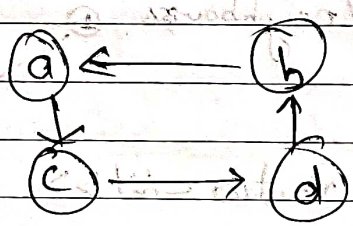
Path → Sequence of Nodes.



Paths.
for a → e:
1. a - b - e.
2. a - c - d - e.
3. a - c - d - b - e
4. a - b - d - e.

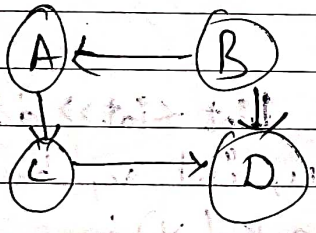
Same concept for directed.

(*) Cyclic Graph. → ~~Undirected~~ / Directed, cyclic
Weighted / Non-Weighted, cyclic



when we are able to reach the starting node again.

(*) Acyclic Graph → ~~undirected~~ / Directed Acyclic
weighted / Non-weighted Acyclic.



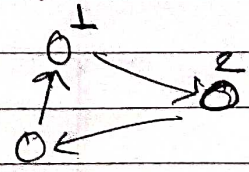
Representation of Graph.

1. Adjacency Matrix
2. Adjacency List.

(1) we get the input in form.

number of nodes.
number of edges

n = 3, m = 3.
0 → 1
1 → 2
2 → 0



Edges list

Page No.

Date ___/___/_____

Adjacency Matrix.

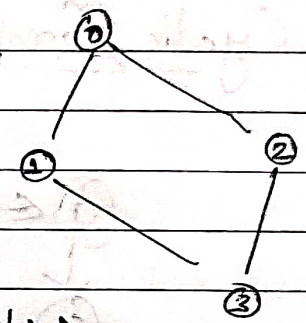
	0	1	2
0	0	1	0
1	0	0	1
2	1	0	0

$O(n^2)$.

(2) Adjacency List → for undirected & directed.

0 → 1, 2.
1 → 0, 3.
2 → 3, 0
3 → 1, 2.

connected nodes / neighbours



implementation

map<int, vector<int>>.

Code:

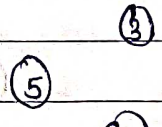
```
class Graph {
    pubic:
        unordered_map<int, list<int>> adj;
    void addEdge (int u, int v, bool directed) {
        adj[u].push_back(v);
        if (directed) {
            adj[v].push_back(u);
        }
    }
    void printGraph() {
        for (const auto& i : adj) {
            cout << i.first << "->";
            for (const auto& j : i.second) {
                cout << j << " ";
            }
        }
    }
};
```

Page No.

Date __/__/____

(1) Null Graph → No edges in the Graph

(2) Trivial Graph → Single vertex & smallest graph possible.

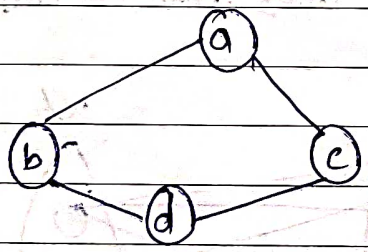


Null graph

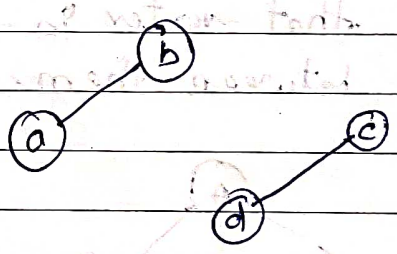
Trivial Graph.

(3) Connected Graph → we can visit any other nodes in a graph with the help of one node.

(4) Disconnected Graph → The graph in which one node is not reachable from a node.



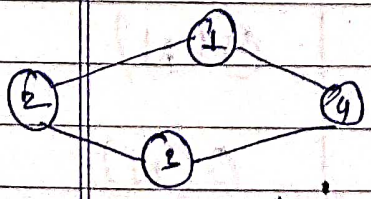
connected



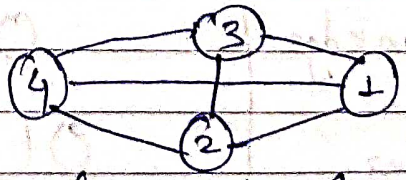
Disconnected.

(5) Regular Graph → The graph in which the degree of every vertex is equal to k is called k regular graph.

(6) Complete Graph → each and every node connected to each other.



2- Regular

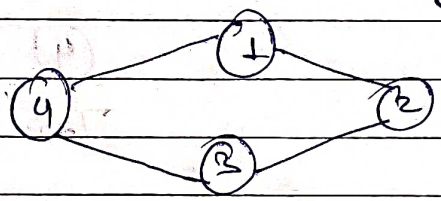


Complete Graph.

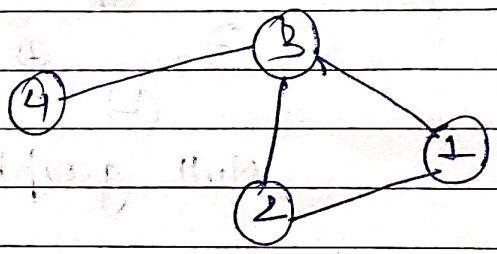
Saathi

Date ___/___/_____

7. Cycle Graph. → The graph in which the graph itself a cycle is called cycle graph. degree each vertex is 2



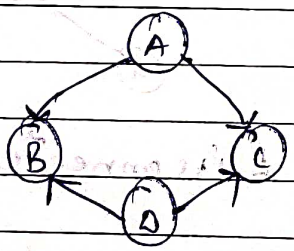
Cycle Graph

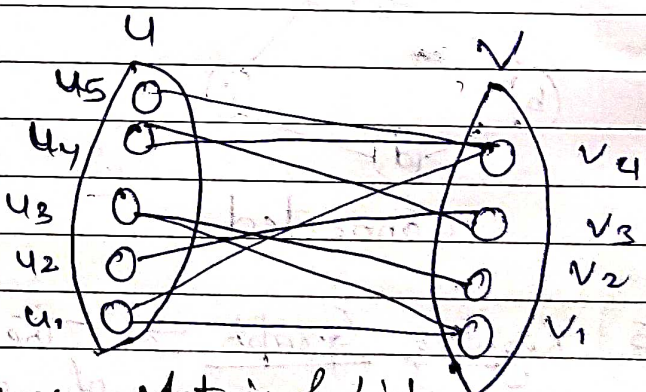


cyclic Graph.

8. Directed Acyclic Graph. → Directed graph which does not contain any cycle.

9. Bipartite Graph → A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.





Comparison between Adjacency Matrix & List

Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing an Edge	O(1)	O(N)
Initializing	O(N*N)	O(N)

Page No.

Date __/__/____

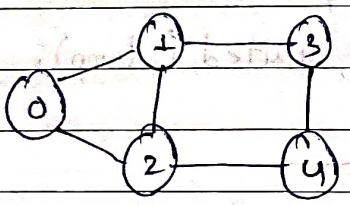
Saathi

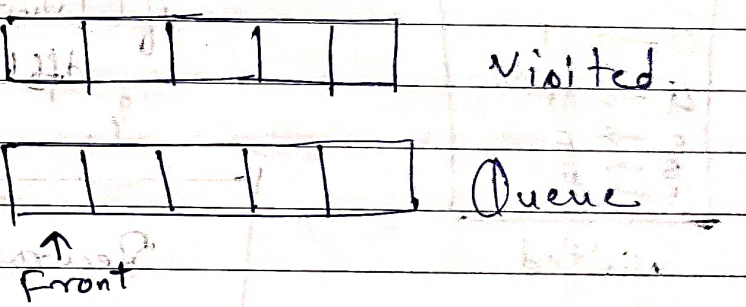
BFS → Breadth first Search & Traversal Technique.

BFS is a Traversal Technique algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level.
It commonly used in path finding connected components and shortest path problems in graph.

§ BFS for a Tree is also known as level order §

We need to keep track on visited & unvisited nodes





(1) 0, —, —, —, —
0, —, —, —, —

(2) 0, 1, 2, —, —
1, 2, —, —, —

(3) 0, 1, 2, 3, —
2, 3, —, —, —

(4) 0, 1, 2, 3, 4.
3, 4, —, —, —

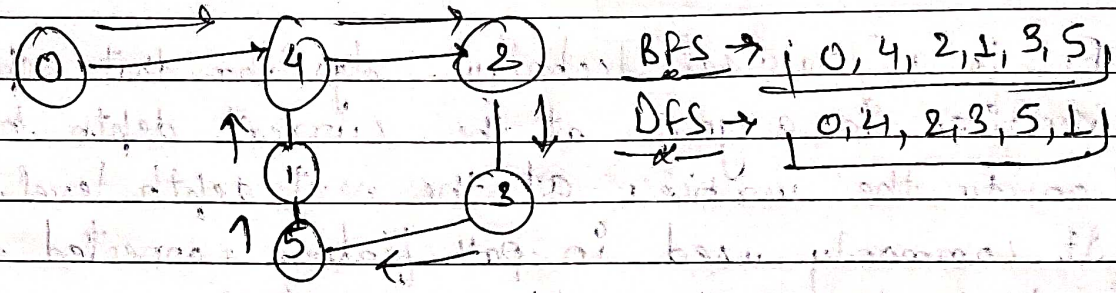
Then in next two passes 3, & 4 will get poped.

T.C. (O(V+E)) S.C → O(V)

Saathi

Date ___/___/_____

DFS → Depth first search.



Algorithm →

Traversing all nodes using for loop.

```
vector<int> temp;
for(int i=0; i<V; i++){
    if(!visited[i]){
        dfs(adjList, visited, i, temp);
    }
}
```

i →	T/F
0 →	T
4 →	T
2 →	T
3 →	T
5 →	T
1 →	T

visited (map)

adjList (int, list<int>)

0 →	2, 4
1 →	5, 3, 4
2 →	4, 3
3 →	5, 2
4 →	1, 2
5 →	1, 3

Recursion Tree {

dfs(0) (1) if(!T) → mark true

dfs(4) if(!T) → mark true

dfs(2)

dfs(3)

dfs(5)

dfs(1)

Page No.

Date __/__/____

Saathi

(i) Cycle Detection :- cycle detection in Undirected Graph.

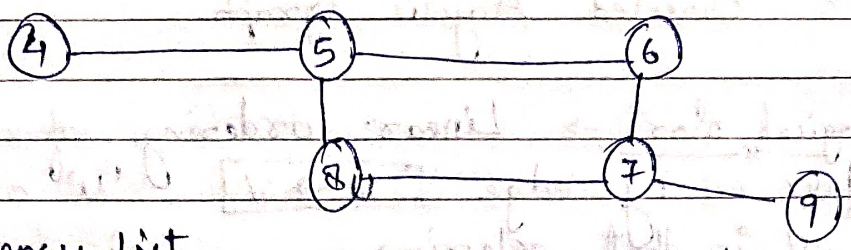


fig 1.1

(1) Adjacency List
(2) Visited table
(3) Parent table

```
if (visited == true && !Parent) {
    return True;
    // cycle present
}
if (!visited[i]) {
    visited[i] = true;
```

(ii) Cycle Detection → Cycle detection in Directed Graph.

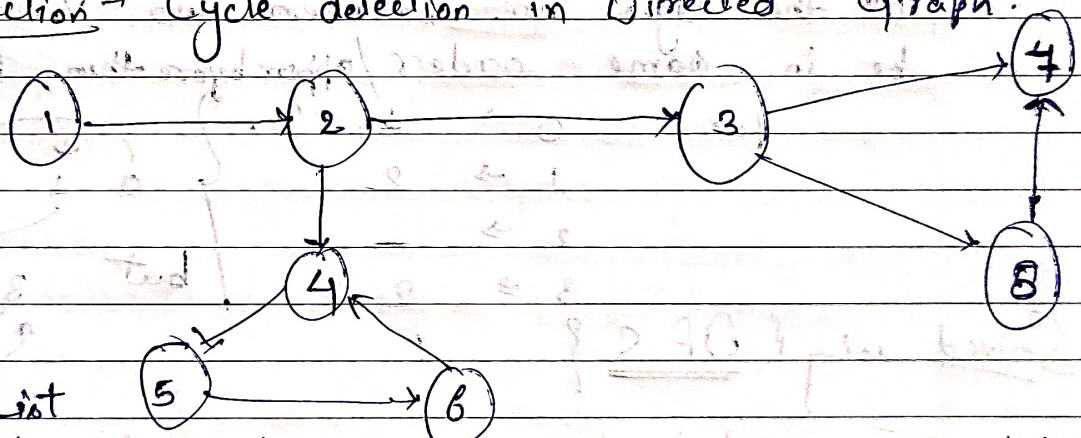


fig 1.2

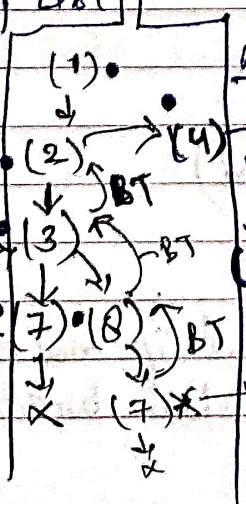
(i) Adjacency List
(ii) dfs visited → Element will be unmarked when backtrack happen
(iii) visited → Mark element in both table (visited)

Example → 1.2

Call stack

Adj List

1 → 2
2 → 3, 4
3 → 7, 8
4 → 5
5 → 6
6 → 4
7 → —
8 → 7



Visited

1	2	3	4	5	6	7	8
↓	↓	↓	↓	↓	↓	↓	↓

Nodes →

1	2	3	4	5	6	7	8
↓	↓	↓	↓	↓	↓	↓	↓

(5) • Dfs visited.
(6) • → (4) → (done table me true) (done true he '4'.
• already visited → cycle present

BT → Backtrack.

Page No.

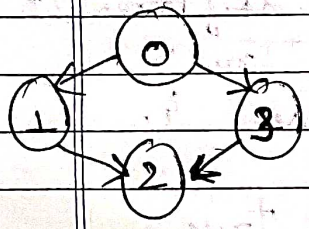
Saathi

Date __/__/____

Topological Sort

DAG → Directed Acyclic Graph.

Topological Sort → Linear ordering of vertices such that for every edge [u → v] 'u' always appears before 'v' in that ordering.



0 1 3 2 → valid

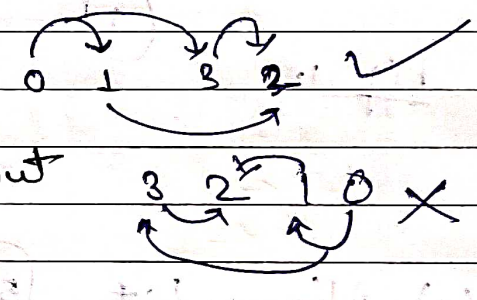
but

3 2 1 0 → Invalid.

Adj List
0 → 1, 3
1 → 2
3 → 2
2 → —

in list 0 1 3 2. according Adj list & Graph all the edges are between two vertex must be in same order / appear before then in sorted way.

for 0 → 1, 3.
1 → 2
2 → —
3 → 2



Solved using { DFS }

Topological Sort cannot be applicable on Cycle graph.

1. Cycle detection.
2. Use Stack { insert at that point where there next propogation is not possible }.
3. Print the stack.

T.C → (O(N+E)), S.C → O(N)

Algo:

(1) Adj List
(2) Call dfs topological Util function & if not visited call fun
(3) Insert the node into stack when backtrack.

Page No.

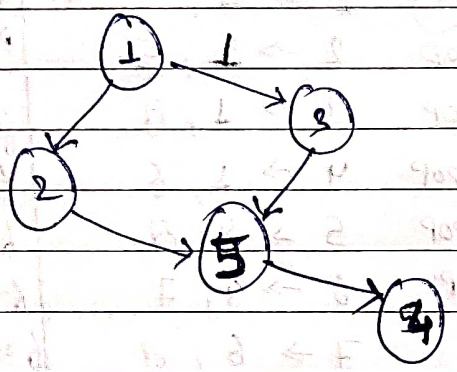
Date __/__/____

Kahn's Algorithm

Topological sort { BFS }

Algorithm

(1) Find indegree of all nodes.
(2) queue → insert all nodes with '0' indegree.
(3) apply BFS.



Indegree data structure

0	1	1	1	2
1	2	3	4	5

queue

0 indegree.

(4) Remove front from queue & add to ans.
(5) When 1 is removed indegree of 2 & 3 will be removed.

updated data st. of Indegree

0	0	0	1	2	✗
1	2	3	4	5	

removed

Repeat :

T.C → O(N+E) { Linear.
S.C → O(N+E)

Cycle Detection. { Directed Graph } (BFS)

Implemented using Kahn's Algorithm.

(1) If there is a valid topological sort then the ans will come for the number of node.

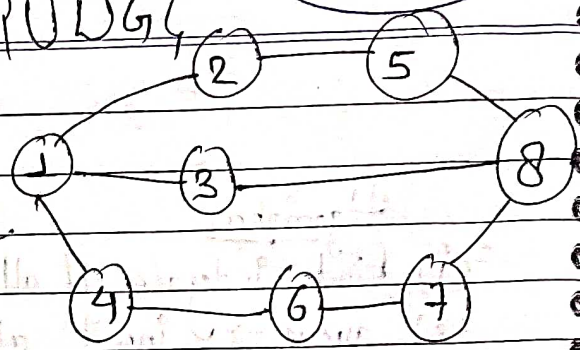
(2) If it is not a valid topological sort the ans will not equal to number of node. { count the node in the loop } { of queue instead of vector }.

Page No.

un-directed graph.

Saathi

Date __/__/____ Shortest Path in [UDG]



* Approach & Algorithm.

using BFS.

(1) Adj List, visited, Track Parents; queue.

Algorithm

	visited	parent	7 so on.	Adj List	queue action
1	✓1	-1	6	(1) 1 → 2, 3, 4	front = 1.
2	✓1	1	~~8~~	push, pop 2 → ~~1~~, 5	Push front neighbours.
3	✓1	1	~~5~~	push, pop 3 → 1, 8	front = 2
4	✓1	1	~~4~~	push, pop 4 → 1, 6	front = 3
5	✓1	2	~~8~~	push, pop 5 → 2, 8	front = 4
6	✓1	4	~~2~~	push, pop 6 → 4, 7	front = 5.
7	✓	6	~~1~~ → pop	7 → 6, 8	front = 8
8	✓1	3	queue	8 → 5, 3, 7.	

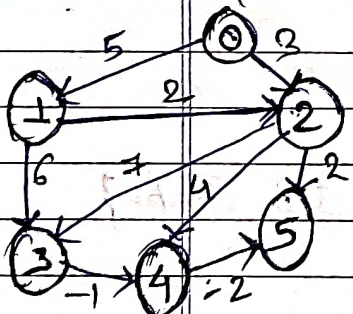
visited array. parent array

Now check Parent array | -1 | 1 | 1 | 1 | 2 | 4 | 8 | 3 |

Shortest path 1 → 8. → parent of 8 → 3 → 1

reverse → 1 → 3 → 8.

Shortest Path in (DAG)



1 → 0, 1 → 1, 1 → 2, 1 → 3, 1 → 4, 1 → 5

{ INF, 0, 2, 6, 5, 3 }

Approach & Algorithm.

(1) Topological Sort. { find order }

(2) Update distance array using order from Toposort

using Topological Sort → (1) Stack, (2) visited Array, (3) Adj List (weighted)

Weighted List

unordered_map

<int, list< pair<int,int> >>

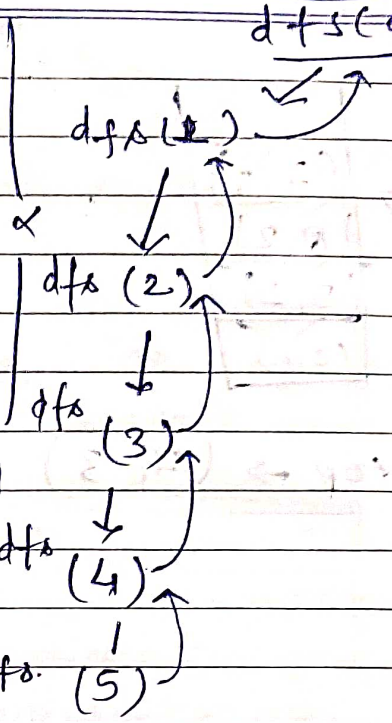
Saathi

Date: __/__/____

dfs(0)

Adj. List →

0 → [1,5], [2,3]
1 → [2,2], [3,6]
2 → [3,7], [4,4], [5,2]
3 → [4,-1]
4 → [5,-2]
5 → X



while backtracking we check the unvisited member & mark them visited.

the insert into stack same as topological sort

Now calculate shortest Path.

dist[src] = 0

∞	0	2	6	5	4
0	1	2	3	4	5

top=(0) → skip. top 1 → 0 → [2,2], [3,6]

top (2) → [3,7], [4,4], [5,2]

2+7=9 but 6<9

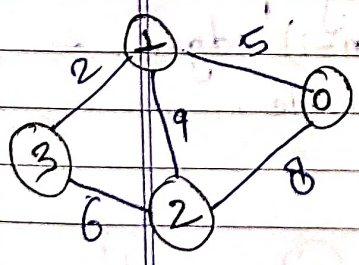
Now use stack & Adj List.

soon.

---

0 → 0 → 0 Dijkstra's Algorithm.

0 → 1 → 5
0 → 2 → 8
0 → 3 → 7
[0, 5, 8, 7]



adj List
0 → [1,5], [2,8]
1 → [0,5] [2,9] [3,6]
2 → [0,8] [1,9] [3,6]
3 → [1,6] [2,6]

Approach → (1) Dist Array → | 0 | 5 | 8 | ∞ |
(0) (1) (2) (3)

(2) Set / Priority Queue

abhi set se karenge.

Now → checking neighbour of '0' → (5,1) & (8,2) (DTO)

repeat

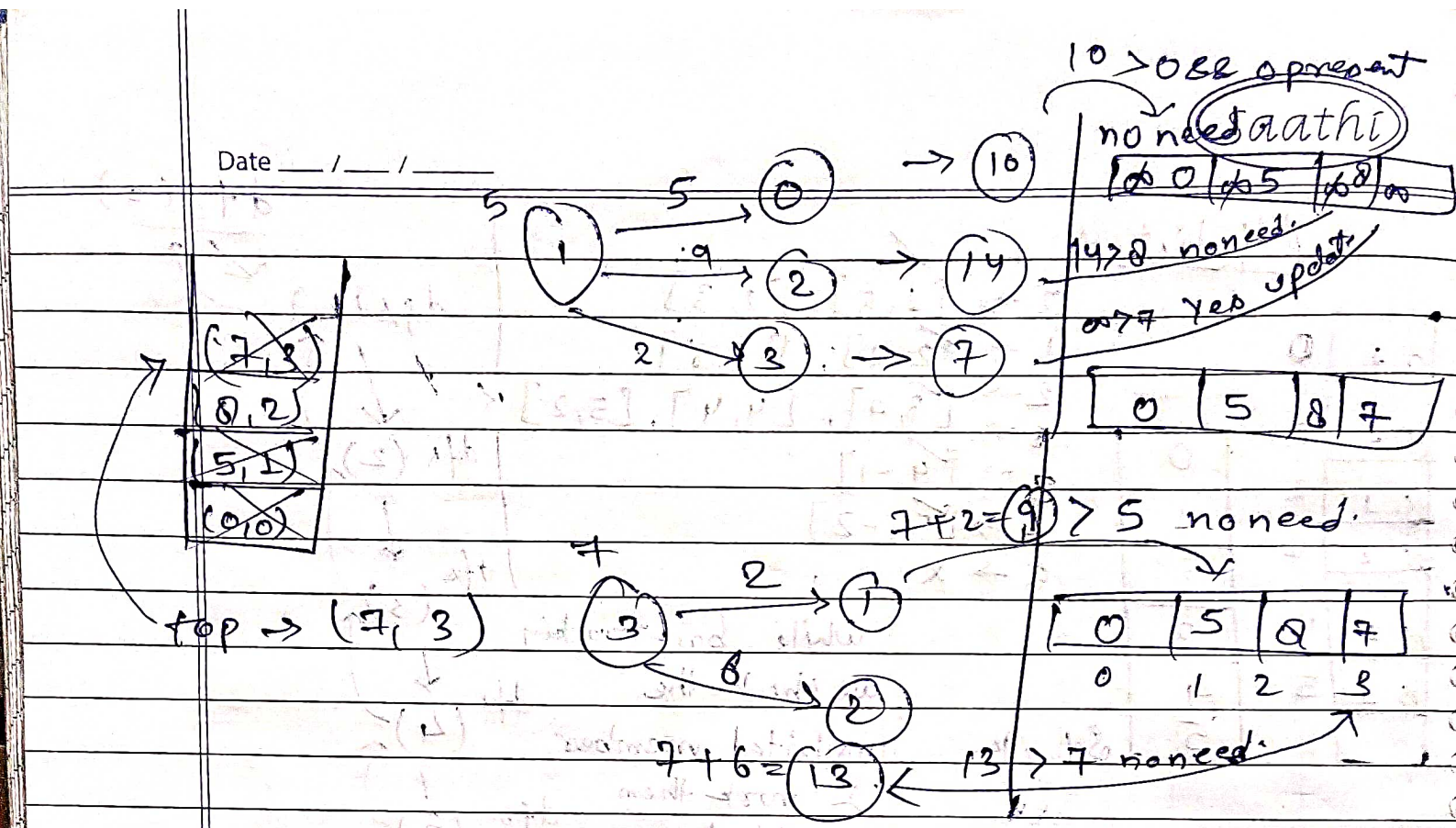
initialize →

(8,2)
(5,1)
(0,0)

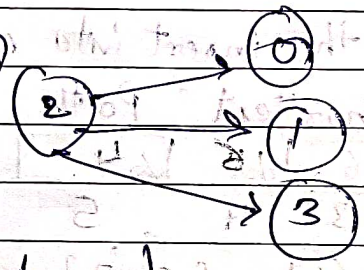
Set <pair<int,int>> <dist, node>

Page No.

Date ___/___/_____



next → (8, 2)



repeat.

T.C → O(E log V)

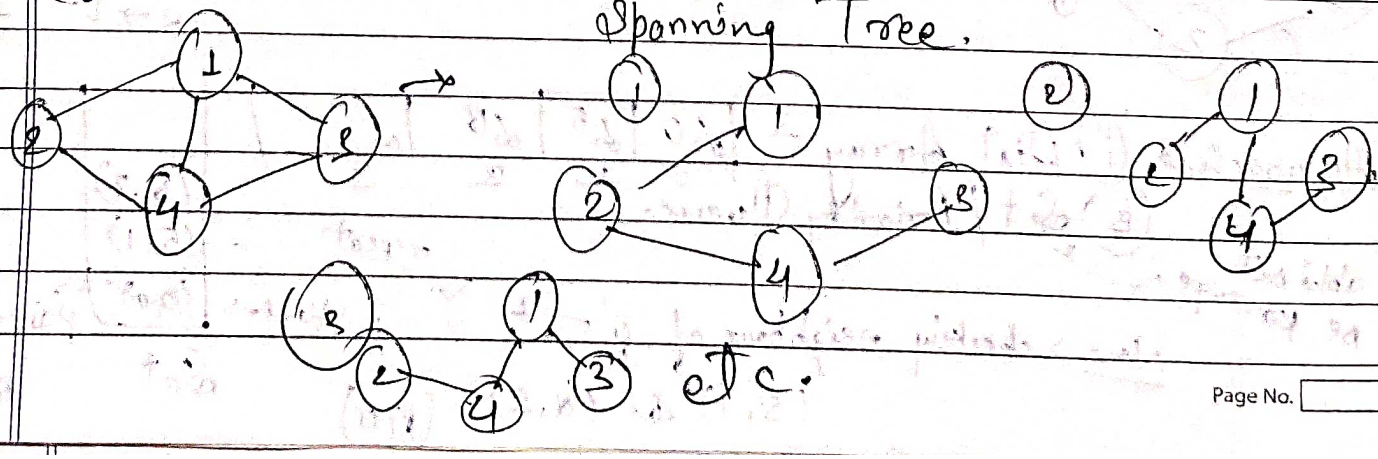
S.C → O(N+E)

To find Minimum Spanning Tree ← Prim's Algorithm.

Spanning Tree → When you convert a graph into trees such that it contains 'N' nodes & (n-1) edges. & every node is reachable from any other node.

* (cycle not present)

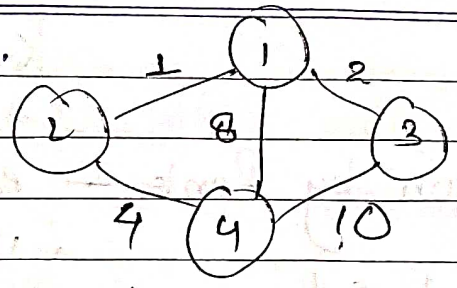
Graph



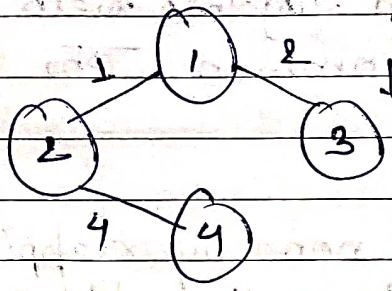
Date ___/___/_____

Saathi

Minimum Spanning Tree →



The spanning Tree which contains minimum cost of weights.



total = 7

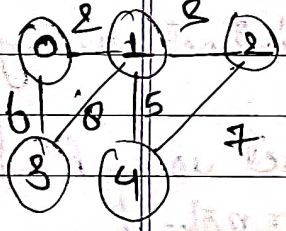
1. Prim's Algorithm.
2. Kruskal's Algorithm.

Data Structure Used.

Prim's Algorithm → (1) Array, (2) Map (Adj List) (3)

(1) Array → key: [0 | ∞ | ∞ | ∞ | ∞] (0 1 2 3 4)

key[0] = 0.
parent[0] = -1



MST: [F | F | F | F | F] (0 1 2 3 4)

Parent: [-1 | 0 | -1 | 0 | -1] (0 1 2 3 4)

(1) Find Min value in key Array.

(2) Mark true in MST Array.

(3) Use Adj List to travel neighbour.

Algo →

mST[i] = false && key[i] < mini

repeat

and in the end we can find the answers using parent Array.

Code is in Vs Code.

Page No:

Saathi

Date __/__/____

Kruskal's Algorithm & Disjoint Set

* Union by Rank & Path Compression ? (Union & Find Algorithm ?)

Disjoint Set

Two sets are called disjoints sets if they don't have any element in common. The intersect of sets is a nullset.

A data structure that stores non-overlapping or disjoint subset of elements is called disjoints set data structure. The disjoint set data structure supports following operation.

1. Adding new sets to disjoint set.
2. Merging disjoint sets to a single disjoint set using union operation.
3. Finding representative of a disjoint set using Find
4. check if two sets are disjoint or not.

Data structure required :-

Array → An array of integer is called Parent[]. If we are dealing with 'N' items. 'i'th element of the Parent[] array is the parent of the 'i'th item. These relationship creates one or more virtual trees.

Tree → It is a Disjoint set. If two elements are in the same tree, then they are in the same disjoint set. The root node (or the topmost node) of each tree is called representative of the set. There is always a single unique representative of each set. A simple rule to identify a representative is if 'i' is the representative of a set, then Parent[i] = i

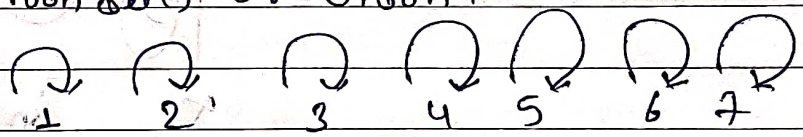
Date ___/___/_____

Saathi

If i is not the representative of his set, then it can be found by traveling up the tree until we find the representative.

Operations on DisJ Set → ① Find ② Union.

Find → FindParent() or FindSet
Union → UnionSet() or Union.



Ex →

Here all 7 nodes are component of graph & have parent of there own if we apply FindParent();

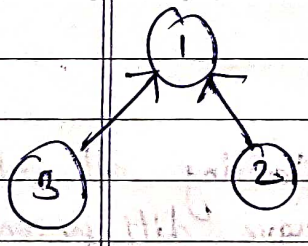
FindParent(1) → 1 for 2 → 2 and so on.
we are performin Union by rank & path compression her

Rank array

X	1	0	0	1	0	1	0
0	1	2	3	4	5	6	7

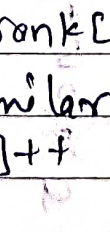
① Union(1,2)
↳ ① parent(1) → 1
parent(2) → 2
check rank = [0,0] (1, 2)
no rule now.
assign '1' as parent of both rank[1]++

② Union(2,3)
parent(2) → 1, parent(3) → 3
check rank[1] > rank[3]. assign
parent[3] = 1;



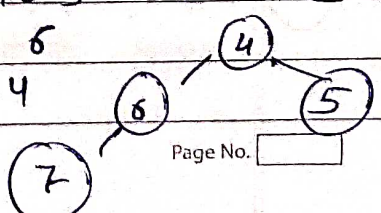
③ Union → (4,5) Same as above
rank[4]++

④ Union(6,7) Similarly
rank[6]++



⑤ Union(5,6) parent[5] = 4
parent[6] = 6

Rank [1], [1] no rule
4 6
parent[6] = 4
rank[4]++

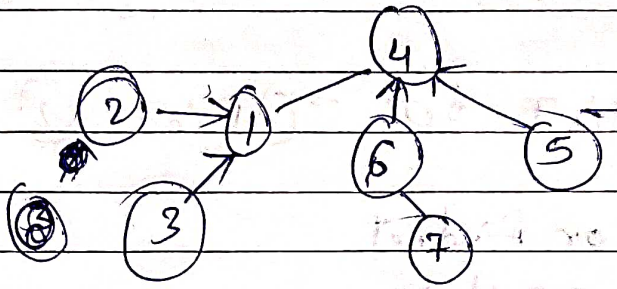


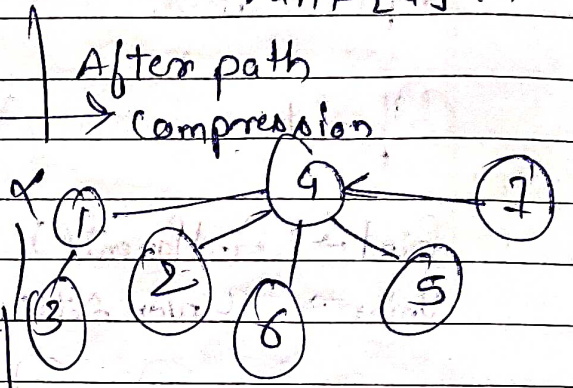
Page No.

Saathi

Date ___/___/_____

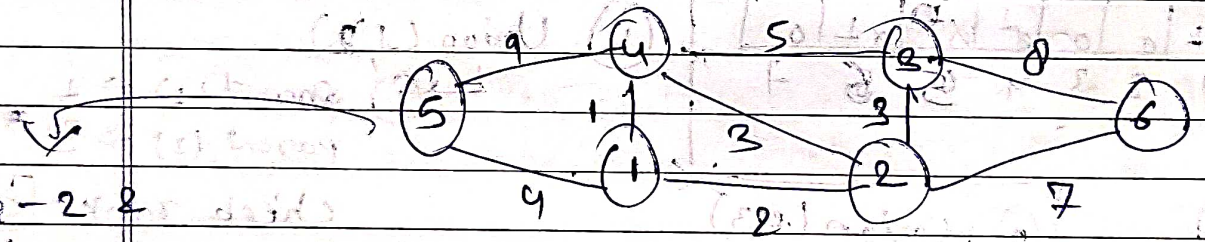
union(3,7) parent[3] → 1 | rank[1] < rank[4]
parent[7] → 4 | parent[1] = 4
rank[4]++





in the end parent is same.

Kruskal's algorithm → (1) adj List ✗ no need.
we need linear data structure
we need to store u, v, w → (weight b/w them)
and then sort them in that data st.

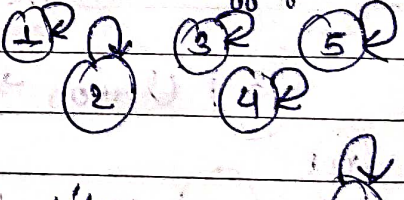


1 - 2 2
1 - 4 1
1 - 5 4
4 - 5 9
4 - 3 5
2 - 4 3
2 - 3 3
2 - 6 7
3 - 6 8

Sorted →

	wt	u	v
1	1	1	4
2	1	2	
3	2	3	
3	2	4	
4	1	5	
5	3	4	
7	2	6	
8	3	6	
9	4	5	

(1) in starting all node have diff. parent



(2) if diff. parent merge them

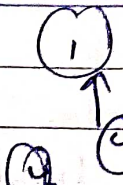
(1) 1, 1, 4 → 1, 4 are in same component.
(1) parent, (2) merge / not merge (add)
parent → same / not same
same → ignore
not same → merge

Page No. ____

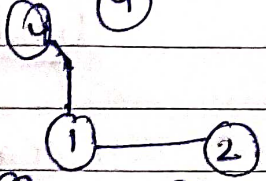
Saathi

Date ___/___/_____

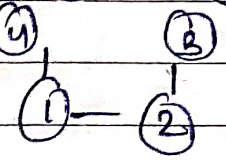
(1) 1, 1, 4 → 1 → parent = 1 merge
4 → parent = 4



(2) 2, 1, 2 → 1 → 1 not same merge
2 → 2

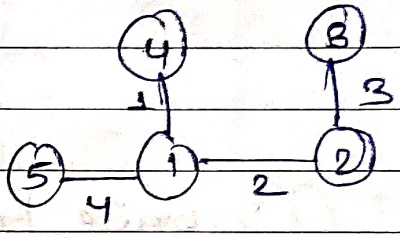


(3) 3, 2, 3 → 2 → 1 not same merge
3 → 3



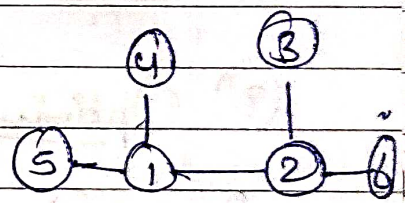
(4) 3, 2, 4 → 2 → 1 same ignore
4 → 1

(5) 4, 1, 5 → 1 → 1 not same merge
5 → 5



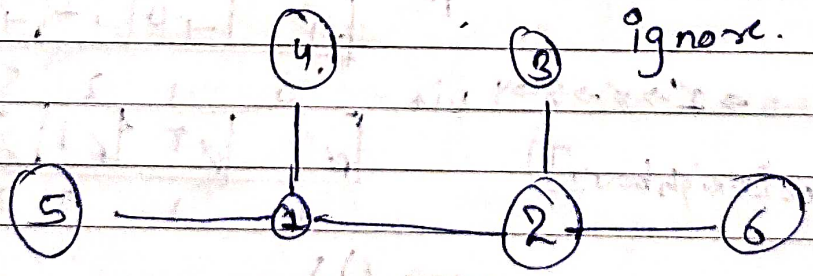
(6) 5, 3, 4 → 3 → 2 → 1 same ignore
4 → 1

(7) 2, 6 → 2 → 1 not same merge
6 → 6



(8) 3, 6 → 3 → 2 → 1 same ~~merge~~ ignore
6 → 2 → 1

(9) 4, 5



ignore.

T.C → O(m log m)
S.C → O(N)

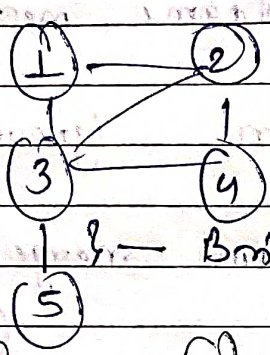
Page No. ____

Saathi

Date __/__/____

Bridges in Graphs

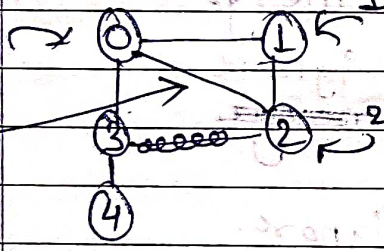
Bridges → If we remove the edges the the component of graph will increase this is called Bridge (That edge is called bridge).



If we remove this edge the component will increase.

(1) **Brute Force** → (1) every edge and remove (2) Try DFS to check connected Comps

(2) **Optimised Way** →



disc 0 1 2 3 4

low 0 0 0 3 4

parent -1 0 1 0 3

vis T T T T T

0 1 2 3 4

(1) low[node] = Timer = 0 → 1 → 2 → 3 → 4

min(low[node], disc[neighbour])

(2) if (neighbour == parent) { ignore }.

(3) while returning → low[node] = min(low[node], low[child])

To check bridge → (low[neighbour] > disc[node])

while returning from (4) the low[4] > disc[3] = Bridge present

Page No.

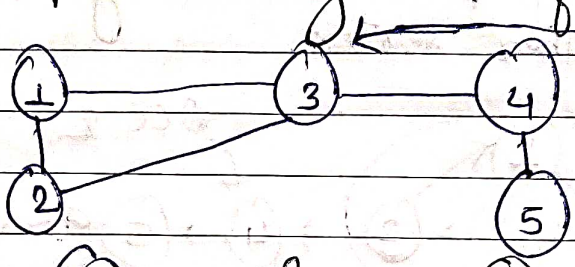
Date __/__/____

Articulation Point

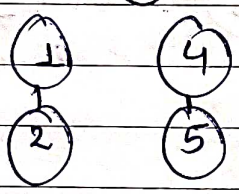
Saathi

An articulation point in a graph is a vertex if removing that vertex will increase the number of connected graph.

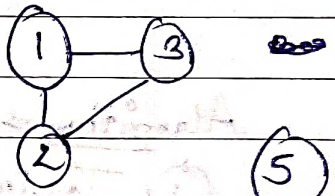
Articulation points represent vulnerabilities in a connected network - single points whose failure would split the n/w into 2 or more components. They are useful for designing reliable n/ws.



Removing 3

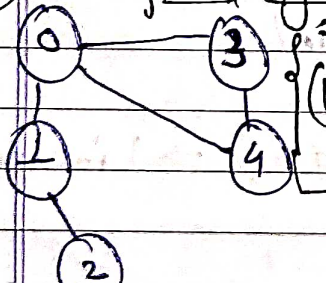


Removing 4



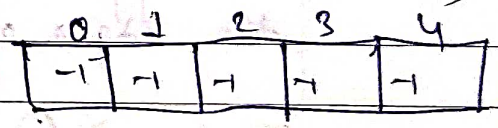
(1) Naive Approach → Removing all the vertex one by one and check it by BFS or DFS if this become disconnected graph. d.s used ? T.C. $O(V*(V+E))$ S.C $O(V+E)$

(2) Tarjan's Algorithm → (optimal) → T.C. $O(V+E)$ S.C $(O(V+E))$

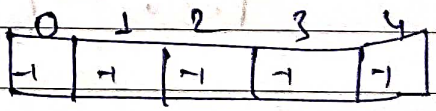


To check AP
(low[nbr] ≥ disc[node] && Parent != -1)

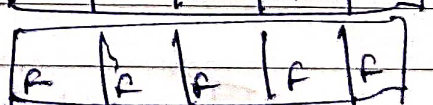
disc



low



parent



(1) timer → 0 →, disc[0], low[0]: → 0

(2) timer → 1 →, disc[3] → 1, low[3] → 1, Parent[3], visited, Vis[T]

(3) timer → 2 → disc[2] → 2, low[4] → 2, Parent[4] → 3, Vis[T]

(4) timer → 3 →

(4) → b.t. → low[node] = min(disc[node], disc[nbr])
if(nbr == parent) ignore

while returning → update parent → low → low[s] = min(low(s), low(nbr))

Now check the AP condition & repeat until nodes get covered.

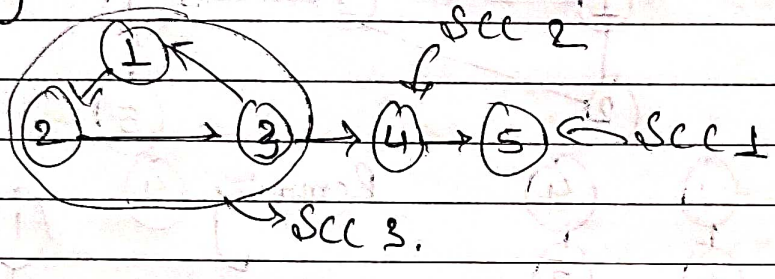
Date __/__/____

Saathi

Kosaraju's Algorithm

This algorithm is used to find / identify strongly connected component in a graph.

Strongly Connected Component → In a graph theory, a strongly connected component (SCC) is a maximal subgraph in a directed graph where every vertex is reachable from every other vertex.



Algorithm →

1. Topological Sort → Sort all nodes on basis of their finish ti
2. Transpose Graph → Reverse all the direction of directed edge
3. dfs → count / print SCC.

* Topology Sort uses stack that's why we have to transpose the edges of graph. (code in VS code)

T.C. → $O(N+E)$ S.C. → $O(N+E)$ → Linear.

Bellman Ford

This algorithm can find shortest path if there exist any negative weight, we can also use this algo to find out negative cycle. If cycle present we cannot find shortest path.

$(n-1)$ → times check this formula → if (dist[u] + wt < dis
{ dist[v] = dist[u
wt }

Date / /

Saathi

$(n-1)^{th}$

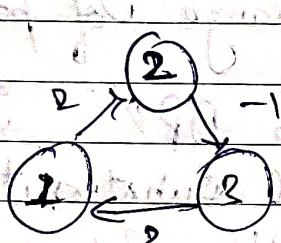
for $(n-1)^{th}$ time we can find the shortest path but in $(n)^{th}$ time we can find if there exist any negative cycle.

(1) $(n-1)^{th}$ → find shortest path.

(2) $(n)^{th}$ → check all distance. if (any distance get update) then there exist negative cycle.

$(n=3)$

(1st)(2nd)



dist[v]

0	∞	3	∞
src → 1	2	3	

edges.

1 → 2 (2)
2 → 3 (-1)
1 → 3 (2)

if (dist[u] + wt < dist[v])

(1) : dist[1] + 2 < ∞ dist[2]

0 + 2 < ∞

2 < ∞

(2) dist[2] + (-1) < dist[3]

2 + (-1) < ∞

1 < ∞

Final ans. will be

0	2	1
1	2	3

Shortest Path.

* n^{th} time to check negative cycle.

Here for all edges & vertex. there is no negative cycles

↓

Graph.

Pseudo code

Date __/__/____

Dynamic Programming

Saathi

DP series.

Dynamic Programming → It is a method used in mathematics and computer science to solve complex problems by breaking them down into simpler sub-problems. By solving only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

How DP work?

1. Identify sub-problems.
2. Store solutions
3. Build up solutions
4. Avoid redundancy.

When use DP?

better than → Greedy

1. Optimal Substructure.
2. Overlapping Sub-problem.
   ↓ recursion

Approach of DP.

1. Top-Down Approach (Memoization): → We start with final solution. Recursion + memoization. & recursively break it down into smaller subproblems.
2. Bottom-up Approach (Tabulation): We start with smallest sub problems & gradually build up to the final solution. We store the results of solved subproblems in a table to avoid redundant calculations.
   - (2a) starts with the smallest subproblems & gradually builds up to final solution.
   - (2b) fills a table with solutions to subproblems in a bottom-up manner
   - (2c) Suitable when the number of subproblems is small & the optimal solution can be directly computed from the solution to smaller subproblems.

Page No.

Saathi

Date __/__/____

(1) Top-down → Fibonnaci Series →.

vector<int> dp(n+1); → for(i=0→n+1){ dp[i]=-1; }

-1	-1	-1	-1	-1
0	- - - -	n	n+1	

```
fib(int n, vector<int> &dp){
    if(n<=1)
        return n;
    if(dp[n] != -1)
        return dp[n];
    dp[n] = fib(n-1) + fib(n-2);
    return dp[n];
}
```

T.C. → O(N)
S.C → O(N)+O(N)
→ O(N)

(2) Tabulation → Bottom-up → ,create dp Array.

(1) Base case check kro. → n==1, n==0.

(2) dp[n] = dp[n-1] + d[n-2];

d[0]=0, d[1]=1

0	1	1	2	3	5
0	1	2	3	4	5

T.C. → O(N)
S.C. → O(N)

(3) Space Optimization → curr = prev1 + prev2

```
prev1 = 0, prev2 = 1;
for(int i=2; i<n; i++){
    int curr = prev1 + prev2;
    prev2 = prev1;
    prev1 = curr;
}
return prev1;
```

T.C. → O(N)
S.C → O(1)

Page No.